CarND Term 1 Project 3 Behavioral Cloning

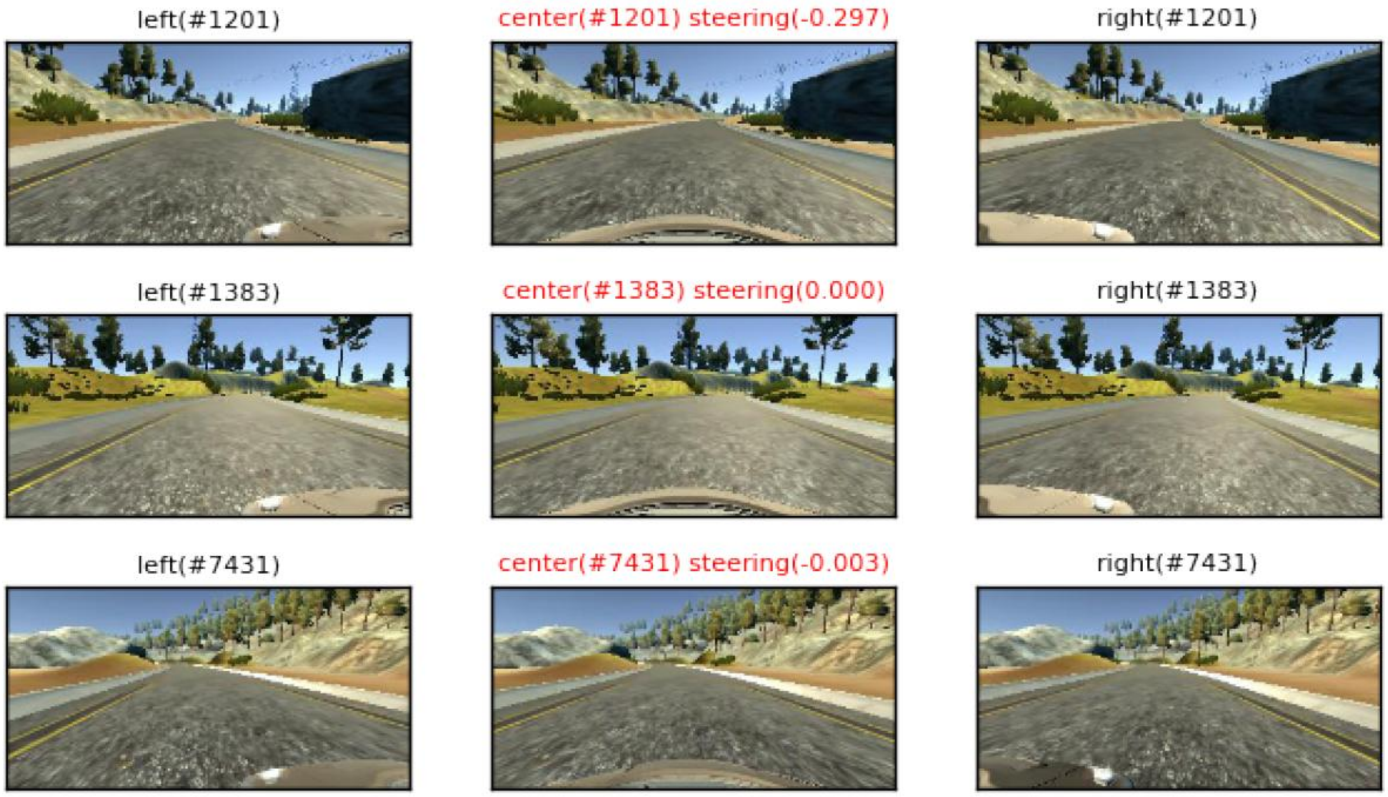
Joe Zhou / ibalpowr@gmail.com

This project, titled as behavioral cloning, is the third project in CarND Term 1. It is a deep learning project, similar to previous project traffic sign classification. Furthermore this project is essentially a *sequential decision making problem* with the goal of throttling and steering a vehicle on a predefined track without derail. My answer to this project is not a solution, but a *hack*, because my answer treats the driving as a regression problem without considering the crucial parameter: time. In theory, my approach will not work indefinitely because over time the buildup of many tiny deviations from the expected trajectory (provided by the convolution neural network) will derail the vehicle. However, this hack works in practices if having sufficient amount of training data and necessary corrections. There are two excellent references: one is Nvidia 2016 "End to End" paper and another is *Berkeley CS294-112 Spring 2017 Lecture 2*.

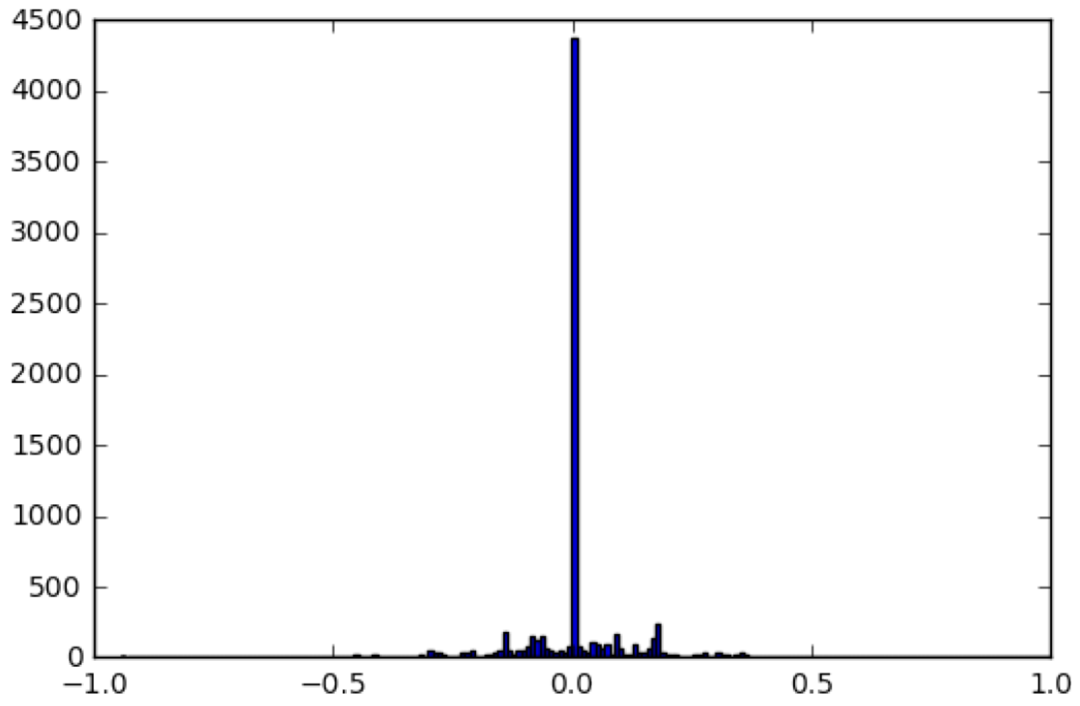
I. Explore and Collect Datasets

Garbage in is garbage out. The first step in deep learning training is always to know what dataset you are using. Getting an effective and sufficient dataset is half of the battle in deep learning.

My default dataset provided by Udacity has two parts: a driving log file and an IMG folder. The log file has a header row of "center, left, right, steering, throttle, brake, speed." The IMG folder has 24108 images which are composed of 8036 images from each of the center, left, and right cameras. In each of 8036 time stamps, center, left, and right cameras take one picture. Below are three random examples of time stamps and their corresponding images from left, center, and right cameras.

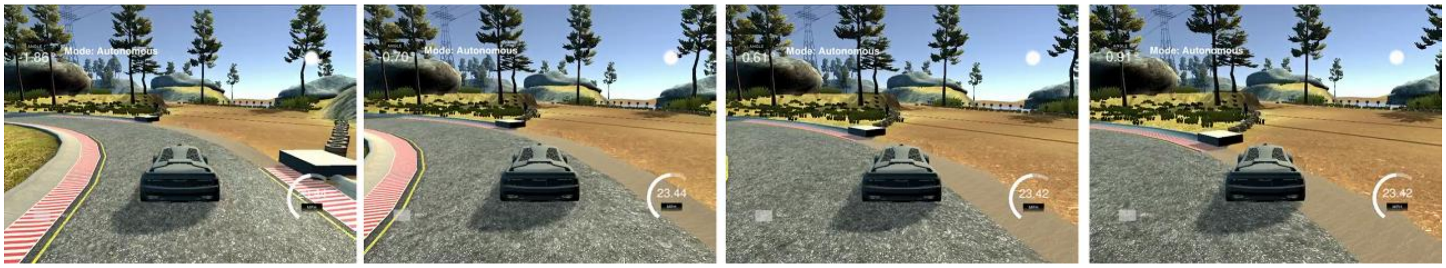

left(#1201)
center(#1201) steering(-0.297)
right(#1201)
left(#1383)
center(#1383) steering(0.000)
right(#1383)
left(#7431)
center(#7431) steering(-0.003)
right(#7431)

Steering has the range of [-1.0, 1.0]. Negative steering angle means the vehicle is making a left turn, positive means making a right turn and 0.0 means going straight. Below is a plot of distribution of the steering angle.

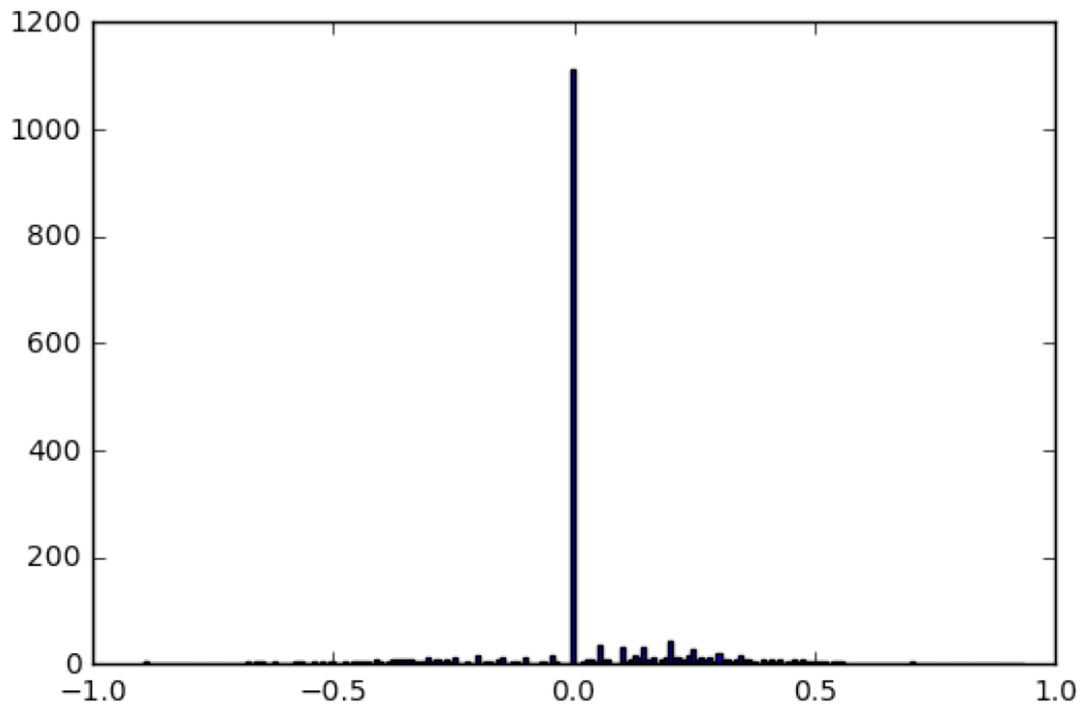


As shown in the plot above, the dominant steering angle is 0.0, going straight. The other three log entries of throttle, brake, and speed are ignored in this project.

During the optimization process, the vehicle cannot make two sharp turns and one sharp turn case is illustrated below.



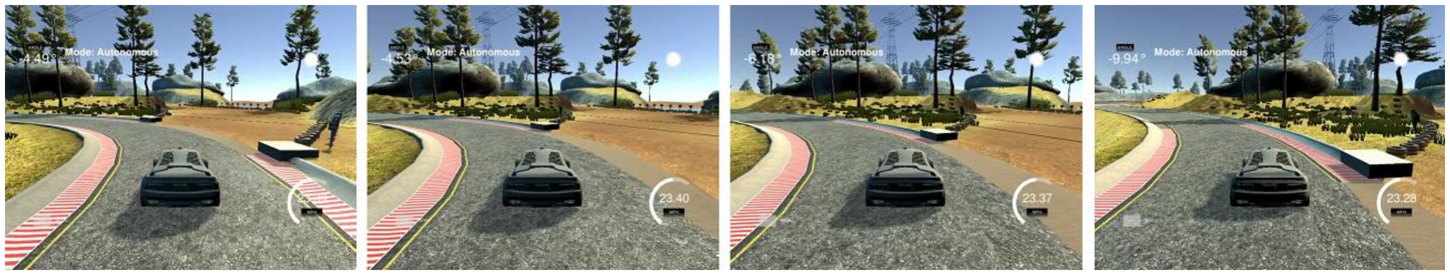
So I have to construct a new dataset by collecting a sequence of corrective actions for those two sharp turns then feed this new dataset into the network to learn and to make the turn successfully. This step of collecting successful actions is similar to one classic computer vision technique: hard-mining false positives. In the collecting process, I set simulator in training mode, drive the vehicle to the position where it derails, stop the vehicle in the *center* of the track, hit recording to start, then use keyboard arrows keys to steer the vehicle to smoothly go around the sharp turn, finally hit stop recording after it completes the sharp turn. Then I back up the vehicle and repeat the same process two more times. Below is distribution of steering angle of the recovery dataset I collect.



The 0.0 steering angles in the recovery dataset are ignored during the training process because those are not the correct actions to take during the sharp turns. Its codes are shown below.

```
# positive means right steering
if (steering > 0.0):
    right.append([center_img, left_img, right_img, steering])

# negative means left steering
elif (steering < 0.0):
    left.append([center_img, left_img, right_img, steering])

else:
    # drop all 0.0 steerings
    if (steering == 0.0):
        None
```

After the learning, now the vehicle can make it through the sharp turn, as illustrated below.



II. Dataset Augmentation and Preprocessing

In my final solution, there is little data augmentation, except flipping the image which is shown below.

```
# flip half of the images
if np.random.randint(2) == 1:
    image, steering = flip_along_y_axis(image,steering)


# 1 mean along y axis ... vertically
# 0 mean along x axis ... horizontally
def flip_along_y_axis(image,steering):
    return cv2.flip(image,1), -steering
```

I did not include the images from “left” and “right” cameras. In my final model, the images from the “center” camera are sufficient to make a successful lap.

In the preprocessing stage, I implement four effective functions: normalization, cropping, darkening, and shuffling. Normalizing the image values from [0, 255] to the range of [-0.5, 0.5] is critical because the regression result for steering is in the range of [-1.0, 1.0]. The prediction and the ground truth should be both centered at 0.0. The normalization is done inside the model with a lambda function as shown below.

```
# normalize to [-0.5, 0.5]
model.add(Lambda(lambda x: x/255.0 - 0.5, input_shape = input_shape))
```

Cropping out the top and bottom portions of the images is also important because the sky on the top and the hood on the bottom are not relevant. And the cropping can speed up the training time because less stuff is being trained. And the code is below.

```
def crop_image(image):

    start = int(image.shape[0] * 0.35)
    end = int(image.shape[0] * 0.875)

    # removes the sky on the top
    # and the hood on the bottom
    new_image = image[start:end, :]

    # resizes to 66 x 220 for nvidia model

    new_image = cv2.resize(new_image, (nvidia_width,nvidia_height), \
                           interpolation=cv2.INTER_AREA)

    return new_image
```

Finally, the other functions include lowering the brightness of the images because there are tree shadows on the track and shuffling the images before feeding them into the network.

III. Model Architecture and Training Strategy

Both Proj2 and Proj3 in Term1 are about deep learning. One key difference between them is the default dataset provided Udacity in Proj3 is *not sufficient* to successfully complete the track no matter what model you choose. Additional data are required. So Proj3 is more complex and advanced involving a new dimensionality on the *interplay* between model selection and data collection.

I started with the model from Nvidia's 2016 End-to-End paper because of its success shown in the Youtube video. However, after many trial and errors, I realized that this model is overkill for this much simpler dataset in Proj3. I have to simplify the model. During the simplification process, I get inspired by the VGG16 [Stanford CS231N Winter 2016 Lecture7]. VGG16 was one of the stop scorers in ImageNet 2014 competition. The most appealing aspect of VGG16 is its simple and uniform architecture, which makes the implementation much easier. I decide to construct a mini-VGG with just 3 layers of convolution and 4 layers of full-connected. As in the original design, my convolution layers all have same set of parameters: filter size of 3x3, zero padding, one stride, and ReLU activation. In order to reduce overfit, I have Maxpooling layers with pool size of 2x2 right after each convolution layer. Maxpooling, a kind of down-sampling, is similar to dropout.

Below is the design sheet for my final model architecture.

design parameters			Term1-Proj3						weights	bias	pixels
number of pixels					b	0	1	c	5,598,960	772	111,404,032
			input volume		256	66	200	3			10,137,600
pad	0	conv1	layer 1	conv-16_3x3	16	3	3	3	432	16	
stride	1										
			activation volume 1		256	64	198	16			51,904,512
		relu1	ReLU function						((image - kernel + pad *2)/stride) + 1		
pool_size	2	pool1	MaxPool function		256	32	99	16			12,976,128
pool_stride	2								((activation - pool)/stride) + 1		
pad	0	conv2	layer 2	conv-36_3x3	36	3	3	16	5,184	36	
stride	1										
			activation volume 2		256	30	97	36			26,818,560
		relu2	ReLU function								
pool_size	2	pool2	MaxPool function		256	15	49	36			6,773,760
pool_stride	2										
pad	0	conv3	layer 3	conv-64_3x3	64	3	3	36	20,736	64	
stride	1										
			activation volume 2		256	13	47	64			10,010,624
		relu3	ReLU function								
pool_size	2	pool2	MaxPool function		256	7	24	64			2,752,512
pool_stride	2										
			flatten function		256			10752			
		fc1	layer 4	fc-512	512			10752	5,505,024	512	
					256			512			131,072
		relu4	ReLU function								
		fc2	layer5	fc-128	128			512	65,536	128	
					256			128			32,768
		relu5	ReLU function								
		fc3	layer6	fc-16	16			128	2,048	16	
					256			16			4,096
		relu6	ReLU function								
		fc4	layer7	fc-1	1			16	16	1	
					256			1			256

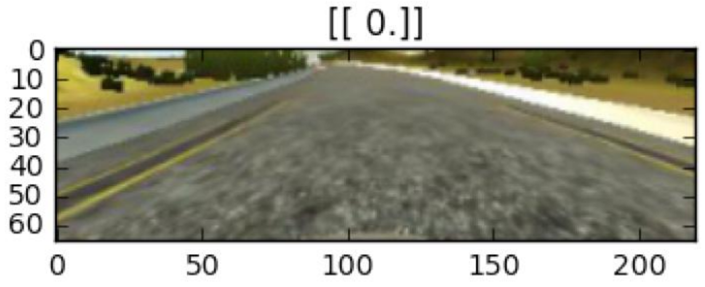
Since deep learning training is a work of both art and science, faster iteration is desired. In the optimization step, the above design sheet helps me trying out different set of parameters much faster. The cells with double underlines are design parameters which can be tuned. The cells in cyan color are the memory usages which can be used to find out the maximum batch size. The cells in black numbers are for sizes of weights/bias files. The correct sizes of pads, strides, and shapes are automatically calculated and propagated among all layers.

In this project I also implement a new technique, data generator, to reduce the consumption of memory usage during training. Data generator is a neat way to pipeline training data into the convolution neural net in real time and place less stress in data storage. And its codes are shown below.

```
def generate_batch_train_from_dataframe(data_df, batch_size):

    batch_images = np.zeros((batch_size, nvidia_height, \
                             nvidia_width, depth))
    batch_steerings = np.zeros(batch_size)

    while True:
        for i in range (batch_size):

            idx = np.random.randint(len(data_df))
            data_row = data_df.iloc[[idx]].reset_index()

            # preprocess image and steering
            img, steering = preprocess_image_train(data_row)

            batch_images[i] = img
            batch_steerings[i] = steering

        yield batch_images, batch_steerings
```

And one example is shown below.

```
# display a test image
for i in range(100):
    next(valid_data_generator)

plt.figure(figsize=(4, 3))
img, steering = next(valid_data_generator)
plt.imshow(img[0])
plt.title(str(steering))
```



Similar to Proj2, I use the standard goto optimizer Adam, settle the learning rate as 1e-4 and split 20% of dataset for validation using SKLearn train_test_split function. Also I use MSE (mean squared error) as a loss function for regression.